Multi-task implementation of multi-periodic synchronous programs

Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, David Lesens

▶ **To cite this version:**

Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, David Lesens. Multi-task implementation of multi-periodic synchronous programs. Discrete Event Dynamic Systems, 2011, 21 (3), pp.307-338. 10.1007/s10626-011-0107-x . inria-00638936

HAL Id: inria-00638936

https://inria.hal.science/inria-00638936

Submitted on 7 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

Multi-task implementation of multi-periodic synchronous programs

Claire Pagetti · Julien Forget · Frédéric Boniol · Mikel Cordovilla · David Lesens

the date of receipt and acceptance should be inserted later

Abstract This article presents a complete scheme for the integration and the development of multi-periodic critical embedded systems. A system is formally specified as a modular and hierarchical assembly of several locally mono-periodic synchronous functions into a globally multi-periodic synchronous system. To support this, we introduce a real-time software architecture description language, named PRELUDE, which is built upon the synchronous languages and which provides a high level of abstraction for describing the functional and the real-time architecture of a multi-periodic control system. A program is translated into a set of real-time tasks that can be executed on a monoprocessor real-time platform with an on-line priority-based scheduler such as Deadline-Monotonic or Earliest-Deadline-First. The compilation is formally proved correct, meaning that the generated code respects the real-time semantics of the original program (respect of periods, deadlines, release dates and precedences) as well as its functional semantics (respect of variable consumption).

Keywords Real-time · Synchronous languages · Preemptive multitasking · Embedded systems.

CR Subject Classification 25: Performance Analysis, 35: General Software Eng & Cybernetics

1 Introduction

1.1 Context

Embedded systems. Digital embedded systems represent a growing part in many manufactured products such as cars, trains, airplanes or medical devices. These systems are re-

Frédéric Boniol · Mikel Cordovilla · Claire Pagetti
ONERA, 2, avenue Edouard Belin, 31055 Toulouse, E-mail: firstname.lastname@onera.fr

Julien Forget
LIFL/INRIA, 40, avenue Halley, 59650 Villeneuve d'Ascq

David Lesens
EADS Astrium Space Transportation, route de Verneuil, 78133 Les Mureaux

Claire Pagetti
IRIT/ENSEEIHT, 2, rue Charles Camichel, 31000 Toulouse

sponsible for various functions such as navigation, guidance, stability, fuel management, air/ground communications. Most of them belong to the so called real-time critical systems family, where failures may have catastrophic consequences.

In this paper we consider multi-periodic control systems, that is to say systems with actions operating at different real-time rates. These systems are generally made up of three kinds of actions: (1) data acquisition (e.g. in an aircraft, at each period the flight control system begins by acquiring the positions of all the flight control surfaces and other flight data such as speeds and accelerations of the aircraft), (2) outputs computation (in response to the inputs acquired at the beginning of the period) and (3) control (sending outputs to actuators).

The complexity of such systems is continuously growing, from a software architecture point of view (more functions to integrate, more communications between functions, more constraints - such as deadline and precedence constraints - to take into account), as well as from an algorithmic point of view (the behaviour of each subfunction, such as a navigation law, becomes more complex). Therefore, designing these systems requires a careful attention and should rely as much as possible on high level languages for both description levels and associated automated code generators.

Concurrent execution. In parallel to this functional evolution, communication and information management technologies evolve to provide new embedded hardware architectures. The implementation of modern embedded systems (whatever the domain - aircraft, cars, trains...) tends to rely on Integrated Modular Architectures (IMA) instead of the more classical federated architectures. In a federated architecture, each functional sub-system is implemented as a sequential code executed on a private computing resource. On the opposite, with IMA, computing resources are shared by several systems (and thus by several design and development teams). To ensure such a genericity, the computing resources are managed by standardized real-time operating systems with multi-tasking capabilities. Thus, the functions are coded by several threads, scheduled concurrently by the real-time operating system. Several norms for real-time operating system are now available such as OSEK (OSEK (2003)), for the automotive domain, and ARINC653 (ARINC (2005)), for the avionic domain.

IMA was designed for numerous expected benefits such as the reduction of the weight and the costs. In terms of real-time, this kind of architecture has an impact on the development: the designer has to ensure that concurrent execution does not jeopardize the functional determinism of the system. The different subfunctions (corresponding to threads) are related by data-dependencies, as data produced by one subfunction may be required by another. Because of the multithreaded execution, the programmer (or the integrator) must carefully control the order in which data is produced and consumed, otherwise the behaviour of the system might not be deterministic and erroneous. This requires to introduce synchronization mechanisms between threads and/or a communication protocol. Synchronizations can be implemented either by precisely controlling the dates at which threads execute so that data is produced and consumed in the right order (*time-triggered* communications with shared memory), or by using blocking synchronization primitives like semaphores (*rendez-vous* communications). The communication protocol indicates to the threads where to write and read data. Programming and implementing such mechanisms manually can be very tedious and error prone. High level software architecture languages, such as AADL (Feiler et al (2006)), GIOTTO (Henzinger et al (2003)) or CLARA (Faucou et al (2004)) for instance, aim at simplifying the specification of the communication and synchronization re-

quirements, and at generating the corresponding low level code automatically. The objective of our work is similar.

Example. To motivate our work, let us consider a simplified flight control system. The role of the system is to control the aircraft altitude, trajectory, speed and auto-pilot. To that purpose, it periodically computes the orders that must be applied to the flight control surfaces in response to the pilot commands and to the aircraft movements. The hardware architecture of the system is made up of a single processor, piloting organs (side-stick, rudder bar, navigation panel), sensors acquiring the aircraft state (GPS for position, accelerometer for speed, etc.) and actuators (ailerons, rudder, etc.). The software architecture is made up of 7 functional tasks and is represented in Figure 1. A fast loop (period 30*ms*) controls the servo-commands. This fast loop is made up of a filter SF (servo-control filter) and a control task SL (servo-control law). The intermediate loop (period 40*ms*) manages the aircraft guidance and is made up of NF (navigation filter) and NL (navigation law). Data required by this intermediate loop is acquired in the fast loop by GNA (guidance and navigation data acquisition). The slowest loop (period 70*ms*) manages the auto-pilot and is made up of GL (guidance law), the inputs of which are acquired and filtered by GF (guidance filter). This loop computes the acceleration to apply. The required position of the airplane is acquired at the slow rate. Each functional task is implemented as a separate real-time task.

If we assume that these 7 tasks are scheduled using a preemptive scheduling policy, it is fairly obvious that the behaviour of the system might vary depending on the scheduling order. Indeed, due to task communications, the task execution order has an impact on the functional behaviour of the system. Therefore, tasks are related by precedence constraints, which are deduced from data freshness requirements.

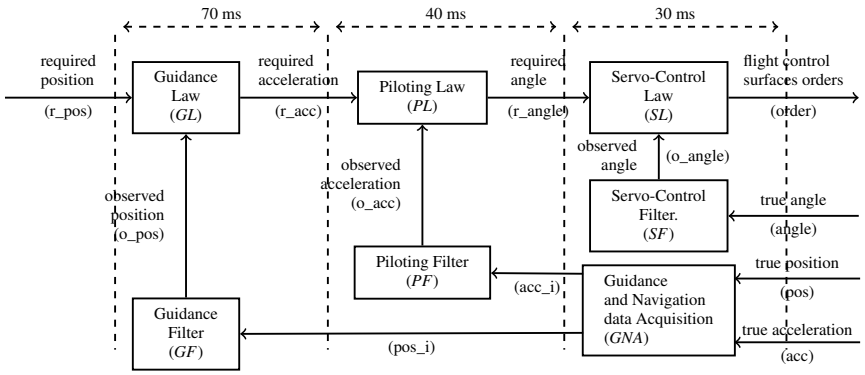

Fig. 1 A simplified flight control system (FCS)

A real size flight control system, may require to integrate up to 5000 tasks on the same computing resource. These tasks may be developed by different persons or different teams from different companies. The integration team then has in charge to assemble them to build the whole system, while respecting all the functional requirements, in particular communication, precedence, and real-time constraints. Due to the huge number of tasks, producing the low level integration code manually, that is to say implementing mechanisms that ensure that the tasks are executed in the right order and that communications take place at the right

dates, is very tedious and error prone. The objective of this paper is to automatize this activity, starting from a formal high level functional description of the system similar to the one presented in Figure 1.

1.2 Related works

Hypothesis Our general goal is to provide a framework for designing safety critical applications. This implies that the designer must prove, mathematically, that the system is correct. In terms of real-time, he/she must prove that the system is predictable which includes that the functions respect their real-time constraints such as period or deadline whatever the configuration. For a multitask implementation, it means that whatever the authorised execution orderings of the tasks and whatever the timing variations (e.g. on the execution time), the behaviour remains valid. To answer this general question, without any additional hypothesis, it requires an exhaustive search among all the executions. For priority-based uniprocessor scheduling, there exist several sufficient tests, exact methods and tools that help proving the correctness. Preemptive on-line scheduling policies are today generally accepted in the safety critical system design. It is the case for IMA where the scheduler proposes sequential code or DM. It is also the case in the space: the ATV (Automate Transfer Vehicle) and Ariane 5 (European heavy space launcher) flight software designs are also implemented on this principle. This is the reason why we focus on classical scheduling policies.

The control / scheduling codesign method. Cervin (2003); Sename et al (2008) proposed a codesign method for flexible real-time control systems, integrating control theory and real-time scheduling theory. Following this codesign method, the scheduler uses feedback from execution-time measures and feedforward from workload changes to adjust the sampling periods of a set of control tasks so that the performance of the controller is optimized. The main advantage is that this achieves higher resource utilization and better control performance (input-output latencies may be reduced). However, this approach assumes that it is possible to derive scheduling requirements at runtime from the quality of the control, and then to change the schedule, also at runtime, depending on values received (inputs) or computed by the system.

The designer must prove that whatever the variations, the system satisfies the real-time constraints. To do this, he/she can reuse a classical result on EDF and fixed priority which are *sustainable* for uniprocessor and synchronous periodic task set (Baruah and Burns (2006)). This means that if a task set is schedulable for one of these policies, it remains schedulable when decreasing execution times, increasing periods or increasing deadlines. If the worst scenario is schedulable, that is when considering the smallest periods, the smallest deadlines and the greatest execution times, then the variations will respect the constraints. However, if this does not hold, the designer must study all the combinations and prove the correctness.

The second drawback is that this approach requires the operating system to be able to modify on-line the real-time attributes of the tasks (which is for instance not possible with Arinc653 or OSEK). The designer must also ensure that such on-line modifications can be done safely and in a bounded time. Furthermore, the overhead due to these on-line modifications should be incorporated in the computation of the response time of the system.

To avoid these difficulties, in our context we suppose that control requirements are translated off-line into *fixed* real-time requirements (periods, deadlines, and precedence constraints).

Matlab/Simulink. SIMULINK (The Mathworks (2009)) is a high-level modelling language widely used in many industrial application domains. In particular, it allows the description of multi-periodic systems in terms of blocks (and subblocks) communicating through data-flows or events. SIMULINK was originally designed for specification and simulation purposes but commercial code generators have been developed since, such as REAL-TIME WORKSHOP EMBEDDED CODER from Mathworks, or TARGETLINK from dSpace.

However, from what we know, SIMULINK has several limitations. Firstly, it allows only harmonic multi-periodic description, which means that the frequency of the blocks must be multiples of one another. So, for instance, the simplified control system presented in Figure 1 cannot be specified with this language. Secondly, it does not allow to define blocks with deadlines different from their periods. Thirdly, synchronization between blocks are implemented by blocking mechanisms (such as semaphores). These mechanisms should be avoided for several reasons. First, the use of semaphores can lead to scheduling anomalies. When used together with semaphores, the EDF and fixed priority scheduling policies, are not sustainable anymore: the system may become unschedulable due to a task that does not take its complete wcet to execute (Graham (1969)).

Another difficulty for using SIMULINK for programming critical systems is that the language and the associated tools lack formal definition. For instance, SIMULINK has several semantics, which depend on user-configurable options and which are only informally defined. Similarly, the conservation of the semantics between the model simulation and the generated code (either for RT WORKSHOP or TARGETLINK) is unclear.

As SIMULINK models are mainly data-flow graphs, a translation from SIMULINK to LUSTRE has been proposed in Tripakis et al (2005). This translation only considers a well-defined subset of SIMULINK models and allows developers to benefit from the formal definition of LUSTRE. This enables formal verification and automated code generation that preserves the semantics of the original SIMULINK model.

Synchronous programming. Synchronous data-flow languages (Benveniste et al (2003)) have successfully been applied to the implementation of reactive systems. They were designed to ensure functional determinism, providing a formal approach to the implementation process. A synchronous program captures the behaviour of a reactive system by describing the computations performed during one basic iteration of the system. Following the synchronous hypothesis, at each iteration the program is considered to react instantaneously, meaning that the duration of a basic iteration (called *instant*) is abstract and that the inputs and outputs are considered to be available simultaneously. In practice, synchronous programs are usually translated to sequential single loop code, implemented by a single task, whose body corresponds to the computations performed during an instant.

Fully sequential compilation is not very well suited for the implementation of multi-periodic systems. Using a real-time operating system with multi-tasking capabilities (and preemption) enables to meet real-time constraints for a much larger class of systems. Therefore, several approaches have been proposed, to translate synchronous programs into a set of communicating tasks scheduled by a RTOS. Among these approaches, Girault and Nicollin (2003); Girault et al (2006) proposed an automatic distribution method of LUSTRE programs into several "computation locations", where different computation location can either correspond to different processors or different tasks. In this approach, the distribution is driven by the clocks of the program. The programmer manually classifies the clocks of the program into different partitions. A program (a task) is then generated for each partition, it contains the computations of all the expressions the clock of which belongs to this partition. Notice that in our approach the partitioning of the program into tasks is automatized.

In SIGNAL (Aubry et al (1996)), a SIGNAL program is translated into a graph representing the clock hierarchy of the program, where each vertex of the graph corresponds to the computations that must be performed for a given clock of the hierarchy and edges correspond to dependencies between the vertices. This graph can then be partitioned into several subgraph, each subgraph is translated into sequential code and encapsulated into a task. In both approaches, tasks are synchronized by blocking communications (semaphores) and thus face the problems already mentioned for SIMULINK. Furthermore, the computation of the priorities of the different tasks is not considered, while it is automatized in our approach.

Finally, the problem of preserving the synchronous semantics of a set of concurrent tasks scheduled with a preemptive policy has been studied by Sofronis et al (2006). The authors proposed a buffering protocol called Dynamic Buffering Protocol similar to ours. The protocol allows to compute a sufficient number of cells[1] required for task communication buffers and determines for each task instance in which cell of the buffer it must write or read. This approach can be applied to periodic as well as aperiodic tasks. Even though this approach addresses a problem similar to ours (transforming a LUSTRE program into a set of tasks managed by a real-time scheduler while preserving the synchronous semantics of the original LUSTRE program), the generation of the tasks from the LUSTRE program and the computation of their real-time attributes, in particular their deadline and priority, is out of their scope.

1.3 Contribution

Forget et al (2008) defined a real-time software Architecture Description Language (ADL) for programming embedded control systems. The language, baptised PRELUDE, transposes the synchronous data-flow semantics to the architecture design level and defines simple real-time primitives to handle multi-rate systems efficiently. The objective of the language is to provide a high level of abstraction for describing the functional and the real-time architecture of a multi-periodic control system. PRELUDE is very close to LUSTRE (Halbwachs (1993)) in its syntax. Being an ADL, it does however serve a different purpose and should be considered as an *integration language* rather than a classic programming language. The development of complex embedded control-systems usually involves several teams, which separately define the different functions of the system. The functions are then assembled by the integrator, who specifies the real-time properties of the functions and the communication schemes between the functions. PRELUDE is designed specifically for the specification of this integration level. In that sense, it can be considered as a real-time software architecture language that enables developers to assemble locally mono-periodic synchronous systems into a globally multi-periodic synchronous system. The second key difference with LUSTRE or similar synchronous languages is that a PRELUDE program can easily be translated into a set of real-time tasks. This leads to better analysis possibilities (performing schedulability analysis) and allows the production of more optimized code (multi-task preemptive scheduling, specific multi-rate communication protocols).

The implementation process is detailed in Figure 2. The program is first specified using the PRELUDE language, briefly presented in Section 2. The objective of this paper is then to describe how, starting from a PRELUDE specification, we can generate a set of real-time threads along with non-blocking communication mechanisms that preserve the PRELUDE semantics. This generation process consists of two steps:

[1] If the tasks are all periodic, the number of cells computed by the protocol is proved minimal.

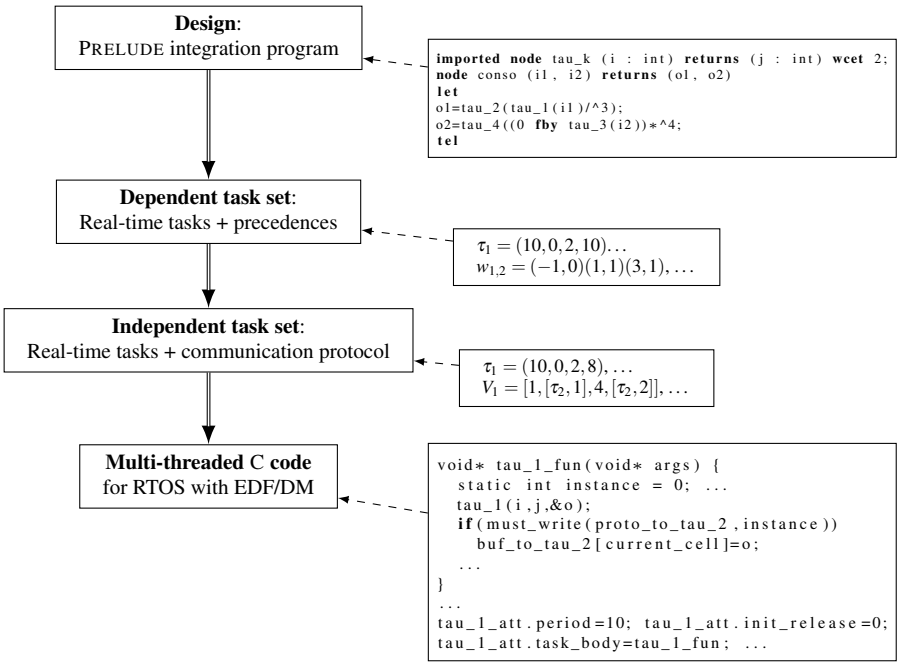

Fig. 2 Compilation chain

1. First we transform the PRELUDE specification into a dependent task set which consists of a set of real-time tasks related by precedence constraints (Section 3). Note that when a precedence constrains two tasks of different periods, it does not constrain all the instances of the two tasks but only some of them. We encode the precedence constraints in periodic *data dependency words* which precisely describe the communication precedences between tasks at the instance level. Due to the deterministic semantics of PRELUDE, this representation of multi-periodic dependencies is very compact.
2. The dependent task set is then translated into an independent task set, i.e. without precedences (Section 4). This consists in two sub-steps:
   (a) The precedence constraints are encoded by adjusting task real-time attributes according to the chosen scheduling policy, EDF or DM (Section 4.1). The encoding of task precedences ensures that for each data-dependency the scheduler will schedule the task producing the data before the task consuming it.
   (b) Then, we rely on *data dependency words* to generate an optimal static non-blocking communication protocol, based on read-write buffer mechanisms, which ensure that task communications respect the synchronous semantics, that is to say, the right instance of the consumer reads the value produced by the right instance of the producer (Section 4.2).
3. The independent task set can then pretty straightforwardly be translated into multi-threaded C code, which is executable with a classic preemptive scheduling policy such as EDF or DM.

Finally, Section 5 gives experimental results concerning the implementation of industrial case studies with our compiler prototype.

2 The PRELUDE Integration Language

The integration of a system is specified using the PRELUDE language defined in Forget et al (2008); Forget (2009). This section provides a detailed description of the language. We first recall some definitions on standard synchronous data-flow languages. We then detail the real-time primitives introduced in PRELUDE to enable the high-level description of a multi-periodic control system. For the most part, these primitives can be considered as particular cases of the classic `when` and `current` LUSTRE operators, or of more recent operators proposed by Smarandache and Le Guernic (1997), Cohen et al (2006). However, the key difference is that real-time characteristics can easily be deduced from our primitives by a compiler, which enables efficient concurrent multi-task implementation of PRELUDE programs. Finally, we present formally the syntax and the semantics of PRELUDE and illustrate it through the programming of the example defined in Figure 1.

2.1 Synchronous Data-Flow

LUSTRE (Halbwachs et al (1991a)), SIGNAL (Benveniste et al (1991)) or LUCID SYNCHRONE (Pouzet (2006)) belong to the family of synchronous data-flow languages. In LUSTRE, the variables and expressions of a program denote infinite sequences of values called *flows*. Each flow is accompanied with a *clock*, which defines the instant during which each value of the flow must be computed. A program consists of a set of equations, structured into *nodes*. The equations of a node define its output flows from its input flows. It is possible to define a node that includes several subnode calls executed at different rates. For instance, suppose that we want to define a simple communication loop, made up of two nodes `F` and `S`, where `S` executes n times slower than `F`. This can be done as follows (here in LUSTRE):

```
node multi_rate(const n: int; i: int) returns(o: int)
var count, vs: int; clockN: bool;
let
  count=0 fby (count + 1);
  clockN=(count mod n=0);
  vs=S(o when clockN);
  o=F(i, current(0 fby vs));
tel
```

The behaviour of the program is illustrated in Figure 3, where we give the value of each flow at each instant. The node `multi_rate` has two inputs: a constant (`n`), which corresponds to the factor relating the rates of `S` and `F`, and an integer (`i`); one output (`o`) and three local variables (`count`), (`vs`) and (`clockN`). The first equation states that at each instant, the value of the flow `count` is computed by the expression `0 fby (count+1)`. The operator `fby` is the *delay* operator, the value of the expression `0 fby (count+1)` is 0 at the first instant *followed by* the previous value of `count` plus 1 at the other instants. Thus `count` is a simple counter. The variable `clockN` is true whenever `count mod n` equals 0, thus every n instants. We use the operator `when` to activate `S` at a slower rate: `o when clockN` takes the value of `o` every time `clockN` is true (the flow is *present*), and has no value otherwise (the flow is *absent*). The operator `when` is an *under-sampling* operator. As a consequence, `S` is executed every n-th instant, and `vs` is also produced every n instants. As `F` executes every instant, `vs` must be *over-sampled* before being consumed by `F`, because the inputs of `F` must be *synchronous*. This is done using the operator `current`, which replaces every absent value due to the `when` operator by the value of the flow the last time it was present.

Designing a multi-periodic system can be realised with such a language. However, readability could be improved, it may not be immediately obvious for a developer reading the

`n`	3	3	3	3	3	...
`i`	i^1	i^2	i^3	i^4	i^5	...
`count=` `0 fby (count+1)`	0	1	2	3	4	...
`count mod n`	0	1	2	0	1	...
`clockN`	true	false	false	true	false	...
`o`	$o^1 = F(i^1,0)$	$o^2 = F(i^2,0)$	$o^3 = F(i^3,0)$	$o^4 = F(i^4,s^1)$	$o^5 = F(i^5,s^1)$	...
`o when clockN`	o^1			o^4		...
`vs`	$s^1 = S(o^1)$			$s^2 = S(o^4)$		...

Fig. 3 Behaviour of a multi-rate LUSTRE/SCADE program

program that "`S` is n times slower than `F`". For more complex periods, the construction of the clocks becomes tedious. More importantly, for the compiler clocks are arbitrary complex Boolean expressions, thus it cannot analyze the program to deduce that "`S` is n times slower than `F`". This effectively prevents an implementation of the program as a set of real-time tasks.

2.2 Synchronous Real-time

PRELUDE adds real-time primitives to the synchronous data-flow model. For this, we modify the synchronous model to relate instants to real-time and we define a small set of high-level real-time primitives based on a specific class of clocks called *strictly periodic clocks*, which represent the real-time rate of a flow.

2.2.1 Relating Instants and Real-time

The synchronous hypothesis states that as long as computations performed during an instant complete before the beginning of the next instant, we can completely abstract from real-time (i.e. from considering at which point of time computations take place). Thus the temporal behaviour of the system is described on a *logical time scale*: we are only interested in the order in which computations are performed inside an instant and in which computations occur at which instant.

PRELUDE follows a *relaxed synchronous hypothesis* (introduced by Curic (2005)), which states that computations must end before their next *activation*, instead of before the next instant (defined by the base clock of the program, i.e. the clock of the inputs of the program). Indeed, a multi-rate system can be considered as a set of locally mono-periodic processes assembled together to form a globally multi-periodic system. Applied to the synchronous model, this means that locally each process has its own logical time scale, where instants are associated with a given real-time duration. When we assemble processes of different rates, we assemble processes of different logical time scales so we use the real-time scale to compare different logical time scales. For instance, in the previous program if we assume a base rate equal to 10ms and take $n = 3$, `F` has a period of 10ms and `S` has a period of 30ms. The execution represented in Figure 4 is correct with respect to the relaxed synchronous hypothesis, while it is not for the simple synchronous one.

Our model is derived from the *Tagged-Signal Model* of Lee and Sangiovanni-Vincentelli (1996). Given a set of *values* $\mathcal{V}$, we define a *flow* as a sequence of pairs $(v_i,t_i)_{i\in\mathbb{N}}$ where v_i is a value in $\mathcal{V}$ and t_i is a tag in $\mathbb{Q}$, such that for all i, $t_i < t_{i+1}$. The clock of a flow is its projection on $\mathbb{Q}$. A tag represents an amount of time elapsed since the beginning of the execution of the program. Two flows are synchronous if they have the same clock (which

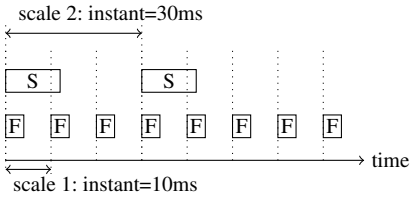

Fig. 4 Logical time scales in a multi-periodic system

implies that the durations of their instants are the same). With this model, we can easily express the relaxed synchronous hypothesis: each flow has its own notion of instant and the duration of the instant t_i is $t_{i+1} - t_i$.

2.2.2 Real-time Primitives

PRELUDE aims at integrating functions that have been programmed in another language. These imported functions must first be declared in the program. The programmer must specify the inputs and the outputs of the function, along with its duration. For instance, the following declaration specifies that the worst case execution time of `S` is 10 time units.

```
imported node S(i:int) returns (o:int) wcet 10;
```

Imported node calls follow the usual data-flow semantics: an imported node cannot start its execution before all its inputs are available and produces all its outputs simultaneously at the end of its execution.

We define rate transition operators that decelerate or accelerate flows. These operators enable the definition of user-specified communication patterns between nodes of different rates. The programmer can over-sample or under-sample a flow periodically as follows:

```
node multi_rate(i: int) returns(o: int)
var vs: int;
let
  vs=S(o/^3);
  o=F(i, (0 fby vs)*^3);
tel
```

The behaviour of the node is equivalent to that of the node defined in LUSTRE in Section 2.1. $e/\hat{}k$ only keeps the first value out of each k successive values of e. On the opposite, $e*\hat{}k$ duplicates each value of e, k times.

Following the data-driven approach of the language, real-time constraints are specified on the inputs/outputs of a node. The rates of the sub-nodes instantiated inside the node are deduced by the compiler from the rates of the node inputs and from the rate transition operators used to implement multi-rate communications. For instance, we can specify the real-time characteristics of the previous example as follows:

```
node multi_rate(i: int rate 10) returns(o: int rate 10 due 8)
```

In this example, `i` and `o` have period 10. Furthermore, `o` has a relative deadline of 8.

2.2.3 Strictly Periodic Clocks

Real-time operators are defined formally using *strictly periodic clocks*. A clock is a sequence of tags and the class of strictly periodic clocks is defined as follows:

Definition 1 (*Strictly periodic clock*). A clock $ck = (t_i)_{i\in\mathbb{N}}$, $t_i \in \mathbb{Q}$, is *strictly periodic* if and only if:

$$\exists n \in \mathbb{Q}^{+*},\ \forall i \in \mathbb{N},\ t_{i+1} - t_i = n$$

n is the *period* of ck, denoted $\pi(ck)$ and $t_0 = 0$.

For simplification, in the following, we will only consider strictly periodic clocks ck the tags of which are in $\mathbb{N}$ (as scheduling theory usually considers dates in $\mathbb{N}$, not in $\mathbb{Q}$). A strictly periodic clock defines the real-time rate of a flow and it is uniquely characterized by its period. Thus, the "strictly periodic clock of period n" will simply be referred to as "clock n".

We then define clock transformations, which produce new strictly periodic clocks by increasing or reducing the rate of an existing strictly periodic clock:

Definition 2 Let α be a strictly periodic clock, operations $/_.$ and $*_.$ are periodic clock transformations, that produce strictly periodic clocks satisfying the following properties:

$$\pi(\alpha /_. k) = k * \pi(\alpha)$$
$$\pi(\alpha *_. k) = \pi(\alpha)/k, k \in \mathbb{N}^\star$$

Strictly periodic clocks are actually a subset of the usual clocks of synchronous languages, which are defined using Boolean activation conditions. However, by restricting to this particular class of clocks, we are able to specify the real-time properties of a system more explicitly and to compile a program efficiently into a set of real-time tasks.

2.3 Syntax and Formal Semantics

2.3.1 Syntax

This section details the syntax of the restriction of PRELUDE we will consider in the rest of the paper. It is close to LUSTRE, however we do not impose to declare types and clocks, which are computed automatically, similarly to LUCID SYNCHRONE. The language grammar is given below:

$$\begin{array}{lll}
cst & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid ... \\
var & ::= & x \mid var, var \\
e & ::= & cst \mid x \mid (e,e) \mid cst\ \texttt{fby}\ e \mid N(e) \mid e/\hat{}k \mid e*\hat{}k \\
eq & ::= & var = e \mid eq; eq \\
typ & ::= & \texttt{int} \mid \texttt{bool} \\
in & ::= & x\ [:\ [typ]\ [\ \texttt{rate}\ n]] \mid in; in \\
out & ::= & x\ [:\ [typ]\ [\ \texttt{rate}\ n]\ [\ \texttt{due}\ n']] \mid out; out \\
decl & ::= & \texttt{node}\ N(in)\ \texttt{returns}\ (out)\ [\ \texttt{var}\ var;]\ \texttt{let}\ eq\ \texttt{tel} \\
 & & \mid\ \texttt{imported node}\ N(in)\ \texttt{returns}\ (out)\ \texttt{wcet}\ n;
\end{array}$$

A program consists of a list of declarations (*decl*). A declaration can either be a node defined in the program (`node`) or a function implemented outside (`imported node`), for instance in C. Node durations must be provided for each imported node, more precisely the worst case execution times (*wcet*).

The clock of an input/output parameter (in/out) can be declared strictly periodic (x : `rate` n, x then has clock (n)) or left unspecified. The type of an input/output parameter can

be integer (*int*), Boolean (*bool*) or unspecified. A deadline constraint can be imposed on output parameters (x : `due` n, the deadline is n, relatively to the beginning of the period of the flow).

The body of a node consists of an optional list of local variables (*var*) and a list of equations (*eq*). Each equation defines the value of one or several variables using an expression on flows ($var = e$). Expressions can be immediate constants (*cst*), variables (x), pairs ((e,e)), initialised delays (cst `fby` e), applications ($N(e)$) or expressions using strictly periodic clocks. $e/\hat{}k$ under-samples e using a periodic clock division and $e*\hat{}k$ over-samples e using a periodic clock multiplication ($k \in \mathbb{N}^*$). Value k must be statically evaluable.

2.3.2 Kahn's semantics

We provide a semantics based on Kahn's semantics on sequences (Kahn (1974)). It is a direct adaptation of the synchronous semantics presented in Colaço and Pouzet (2003) to the Tagged-Signal model. For any operator $\diamond$, $\diamond^{\#}(s_1,...,s_n) = s'$ means that the operator $\diamond$ applied to sequences s_1, ..., s_n produces the sequence s'. Term $(v,t).s$ denotes the flow whose head has value v and tag t and whose tail is sequence s. Operator semantics is defined inductively on the argument sequences. We omit rules on empty flows, which all return an empty flow. In the following, if e is an expression such that $e^{\#} = (v_i,t_i)_{i\in\mathbb{N}}$, we will write e^i to denote v_i, i.e. the i^{th} value produced by e. The semantics of the operators of PRELUDE is given in Figure 5.

$$
\begin{array}{lcl}
\texttt{fby}^{\#}(v,(v',t).s) & = & (v,t).\,\texttt{fby}^{\#}(v',s) \\
\tau^{\#}((v,t).s) & = & (\tau(v),t).\tau^{\#}(s) \text{(where } \tau \text{ is an imported node)} \\
\hat{}^{\#}((v,t).s,k) & = & \prod_{i=1}^{k}(v,t_i').\hat{}^{\#}(s,k) \text{(where } t_1' = t \text{ and } t_{i+1}' - t_i' = \pi(s)/k) \\
/\hat{}^{\#}(s,k) & = & /\hat{}'^{\#}(1,s,k) \\
/\hat{}'^{\#}(n,(v,t).s,k) & = & \begin{cases} (v,t)./\hat{}^{\#}(k,s,k) & \text{if } n = 1 \\ /\hat{}^{\#}(n-1,s,k) & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 5 Kahn's semantics

- x `fby` y concatenates the head of x to y, delaying the values of y by one tag;
- for an imported node τ, $\tau(x)$ applies τ to each value of x;
- $x*\hat{}k$ produces a flow k times faster than x. Each value of x is duplicated k times in the result. The time interval between two successive duplicated values is k times shorter than the interval between two successive values in x;
- $x/\hat{}k$ produces a flow k times slower than x, dropping parts of the values of x.

To complete the definition of the semantics, we need to define how a program behaves when flow values (i.e. sequences of pairs (v_i,t_i)) are assigned to flow variables. To this intent, we defined a denotational semantics (Reynolds (1998)) for the rest of the constructions of the language. This denotational semantics is fairly standard and can be found in Forget et al (2008).

Example 1 Let us illustrate the Kahn's semantics and in particular the data dependencies through a quite simple PRELUDE program:

```
imported node tau_k (i : int) returns (j : int) wcet 2;
node conso (i1, i2, i3,  i4) returns (o1, o2, o3,o4)
let
o1=tau_1(i1);
o2=0 fby tau_2(i2);
o3= tau_3(i3)*^4;
o4= tau_4(i4)/^3;
tel
```

For the first equation `o_1=tau_1(i1)`, if the input sequence of values is $(i_1^1,t_1),(i_1^2,t_2),\ldots$, we obtain the output sequence of values $(o_1^1,t_1'),(o_1^2,t_2'),\ldots$ such that:

$$(o_1^1,t_1'),(o_1^2,t_2'),\ldots = tau\_1^{\#}((i_1^1,t_1),(i_1^2,t_2),\ldots)$$

According to the semantics, it is equivalent to:

$$\forall n \in \mathbb{N}^*, \begin{cases} o_1^n = tau\_1(i_1^n) \\ t_n' = t_n \end{cases}$$

In the same way, we can deduce the relation between the sequences involved in the other equations.

$o_2 = 0$ fby $tau\_2(i2)$	$o_3 = tau\_3(i_3)*\hat{}4$	$o_4 = tau\_4(i_4)/\hat{}3$
$\begin{cases} o_2^1 = 0 \\ \forall n \in \mathbb{N}^*, o_2^{n+1} = tau\_2(i_2^n) \end{cases}$	$\forall n \in \mathbb{N}^*, o_3^n = tau\_3(i_3^{\lceil \frac{n}{4} \rceil})$	$\forall n \in \mathbb{N}^*, o_4^n = tau\_4(i_4^{3(n-1)+1})$

2.4 Example

We are now able to define the PRELUDE program for the system of Figure 1. Each operation (laws, acquisitions, filters) is first declared as an imported node:

```
imported node SF(i: int) returns (o: int) wcet 5;
imported node SL(i1,i2: int) returns (o: int) wcet 5;
...
```

We then describe the communication between the multi-periodic nodes. The program is defined as a set of modular and hierarchical nodes. There are several possible implementations, depending on the communication patterns the programmer wants to implement for communicating nodes. We propose the following assembly solution.

```
node fcs (angle, acc, pos : rate (30); r_pos: rate (70))
returns (ordre)
var r_acc, r_angle, pos_i, acc_i;
let
  (pos_i,acc_i) = GNA(pos,acc);
   ordre = SL(SF(angle),  ((0 fby r_angle)*^4)/^3);
   r_angle = PL (PF(acc_i*^3/^4),((0 fby r_acc)*^7)/^4);
   r_acc = GL(GF(pos_i*^3/^7),r_pos);
tel
```

3 Translation into a Dependent Task Set

This section details how the PRELUDE compiler translates a program into a *dependent task set*, that is to say a set of tasks related by data-dependencies. The tasks correspond to the imported nodes. The main contribution of this section is to define a compact data structure, called *data dependency word* , that represents the data dependency imposed by the semantics between two tasks.

3.1 Data Dependency Words

The PRELUDE semantics gives the rules for computing output flows from input flows. The semantics, detailed in Section 2.3.2, applies on inputs described as an infinite sequence of pairs *(value,tag)* and provides outputs also represented by an infinite sequence of such pairs. The clock calculus computes the clock, and thus the tags, of each flow of the program.

In this section, we describe how to compute the sequences of values of the flows using *data dependency words* . A data dependency word explains how to construct the sequence of outputs values o from sequence of input values i for the expression $o = oprs(i)$ where *oprs* is a combination of the three operators $/\hat{}$, $*\hat{}$ an `fby` . As we only consider multi-periodic systems, data dependency words are finite words.

To simplify the presentation, let us define the set of terms t over PRELUDE operators as follows:

$$\begin{array}{lll} opr & ::= & \texttt{fby} \mid /\hat{}k \mid *\hat{}k \\ t & ::= & opr \mid t.t \mid opr^n \mid t^* \mid (t|t) \end{array}$$

- The sequence $oprs = opr_1.opr_2.\cdots.opr_n$ denotes the composition $opr_n \circ \cdots \circ opr_2 \circ opr_1$, i.e. $oprs(i) = opr_n(...(opr_2(opr_1(i))))$
- opr^n corresponds to n compositions of the operator opr;
- t^* corresponds to an arbitrary number of repetitions of t (possibly 0);
- $(t_1|t_2)$ corresponds to either t_1 or t_2.

In the following we assume that only operator combinations of the following form are allowed: $\texttt{fby}^*.(/\hat{}k|*\hat{}k)^*$ (i.e. `fby` operators are applied first). This constraint does not seem too restrictive in practice, as in most cases programs can easily be rewritten to fulfill this requirement, and significantly simplifies the precedence encoding presented in Section 4.1[2].

Definition 3 (Data dependency word) A data dependency word w is defined by the following grammar:

$$\begin{array}{lll} w & ::= & (-1,d_0).(k,d).u \\ u & ::= & (k,d)|u.u \end{array}$$

with $d_0 \in \mathbb{N}$, $k,d \in \mathbb{N}^\star$.

Let i a sequence of values and $w = (-1,d_0)(k_1,d_1)(k_2,d_2)\ldots(k_n,d_n)$ a data dependency word. The application of w to i produces a sequence of values $o = w(i)$ defined as follows, $\forall p \in \mathbb{N}^\star$:

$$o^p = \begin{cases} init & \text{if } p \in [1,d_0] \\ i^{k_1} & \text{if } p \in [d_0+1, d_0+d_1] \\ i^{k_1+k_2} & \text{if } p \in [d_0+d_1+1, d_0+d_1+d_2] \\ \ldots & \\ i^{q\cdot(\Sigma_{j\in[2,n]}\, k_j)+\Sigma_{j\le l}\, k_j} & \text{if } p \in \\ \qquad [q\cdot(\Sigma_{j\in[2,n]}\, d_j)+\Sigma_{j\le l-1}\, d_j+1, q\cdot(\Sigma_{j\in[2,n]}\, d_j)+\Sigma_{j\le l}\, d_j] & \text{with } q\in\mathbb{N}, l\in[1,n] \end{cases}$$

The first two letters $(-1,d_0)(k_1,d_1)$ correspond to the prefix of the word: the d_0 first values of o equal the initial value (*init*), k_1 is the index of the first value of i actually used to produce o and d_1 tells how many values of o equal this value i^{k_1}. Then, the sequence $(k_2,d_2)\ldots(k_n,d_n)$ is the repetitive pattern of the word: the d_2 next values of o equal $i^{k_1+k_2}$, then the d_3 next values of o equal $i^{k_1+k_2+k_3}$. The pattern is repetitive, thus when we reach

[2] Thanks to this constraint, deadlines adjustment can be performed by a simple topological sort, instead of requiring the computation of a fixed-point. See Forget (2009) Section 10.2.3 for details.

the last pair of the pattern $((k_n,d_n))$, we loop back to the first pair of the pattern $((k_2,d_2))$. So, after using the value $i^{k_1+k_2+...+k_n}$, d_n times, the d_2 next values of o equal $i^{k_1+k_2+...+k_n+k_2}$, then the d_3 next values of o equal $i^{k_1+k_2+...+k_n+k_2+k_3}$, and so on.

Example 2 Let $w_1 = (-1,0)(1,1)(1,1)$ and $o = w_1(i)$. This is the simplest data dependency word. If we apply the previous definition, we obtain: $o^1 = i^1$ since $d_0 = 0, d_1 = 1, k_1 = 1$; $o^2 = i^2$ since $d_2 = 1, k_2 = 1$. Thus, $o^p = i^p$ with $p \in \mathbb{N}^\star$.

If $o_1 = w_1(tau\_1(i_1))$. We obtain $\forall p \in \mathbb{N}^\star$

$$o_1^p = tau\_1(i_1)^p = tau\_1(i_1^p)$$

w_1 is the word computed for the PRELUDE expression $o_1 = tau\_1(i_1)$.

Let $w_2 = (-1,1)(1,1)(1,1)$ and $o_2 = w_2(tau\_2(i_2))$, we have

$$\begin{cases} o_2^1 = 0 \textit{ since } d_0 = 1 \\ o_2^{p+1} = tau\_2(i_2^p) \textit{ with } p \in \mathbb{N}^\star \end{cases}$$

w_2 is the word computed for the PRELUDE expression $o_2 = 0$ `fby` $tau\_2(i_2)$.

Let $w_3 = (-1,0)(1,4)(1,4)$ and $o_3 = w_3(tau\_3(i_3))$, we have $o_3^1 = tau\_3(i_3^1)$, $o_3^2 = tau\_3(i_3^1)$, $o_3^3 = tau\_3(i_3^1)$, $o_3^4 = tau\_3(i_3^1)$, $o_3^5 = tau\_3(i_3^2)$, ... this can be summarized as

$$o_3^p = tau\_3(i_3^{\lceil \frac{p}{4} \rceil}) \textit{ with } p \in \mathbb{N}^\star$$

w_3 is the word computed for the PRELUDE expression $o_3 = tau\_3(i_3) *\hat{}\, 4$.

Let $w_4 = (-1,0)(1,1)(3,1)$ and $o_4 = w_4(tau\_4(i_4))$, we have $o_4^1 = tau\_4(i_4^1)$, $o_4^2 = tau\_4(i_4^4)$, ... this can be summarized as

$$o_4^p = tau\_4(i_4^{3(p-1)+1}) \textit{ with } p \in \mathbb{N}^\star$$

w_4 is the word computed for the PRELUDE to the expression $o_4 = tau\_4(i_4)/\hat{}\, 3$. Notice that this exactly corresponds to the semantics of the PRELUDE program given in the example 1 p. 12.

This example illustrates that the sequence of values produced by the application of a single rate transition operator can be represented by a data dependency word. This can be generalized for complex combinations of rate transition operators.

Proposition 1 *Let oprs be of the form* `fby` $^*.(/\hat{}\ k|*\hat{}\ k)^*$. *For any sequence of values i, the sequence of values produced by* $o = oprs(i)$ *can be represented as* $o = w(i)$, *where w is a data dependency word the length of which is less than* $\prod_{(/\hat{}\ k_i)\in oprs} k_i + 2$.

Proof For a word $w = (-1,d_0)(j_1,d_1)(j_2,d_2)...(j_n,d_n)$, let $l(w) = n-1$ denote the length of w (we do not take into account the prefix of constant length 2) and let $nd(w) = \Sigma_{2\leq i\leq n}\, d_i$. The proof is done by induction on the sequence of operators *oprs*. The base of the induction is when *oprs*$= []$, in which case the word is $(-1,0)(1,1)(1,1)$ (see example above) with $l(w) = 1 \leq \prod_\emptyset k_i = 1$ and $nd(w) = 1$.

Now, for the induction step, let us assume that the word associated to *oprs* is $w = (-1,d_0)(j_1,d_1)(j_2,d_2)...(j_n,d_n)$. We compute the word w' for $oprs.op$. In the following, if w is the word computed for a sequence of operators *oprs* and if *op* is an operator, $w' = op(w)$ denotes the data dependency word that corresponds to the sequence of operators $oprs.op$.

1. if *op*= `fby` , then *oprs* is necessarily of the form `fby` ^{d_0} because of the assumption to apply the delays at the beginning. It entails that $w = (-1,d_0)(1,1)(1,1)$ and $w' =$ `fby` $(w) = (-1,d_0+1)(1,1)(1,1)$ with $l(w') = l(w) = 1$ and $nd(w') = 1$;
2. if $op = \circledast\hat{}\,k$, then $w' = \circledast\hat{}\,k(w) = (-1,k\cdot d_0)(j_1,k\cdot d_1)(j_2,k\cdot d_2)...(j_n,k\cdot d_n)$. Indeed, the flow obtained by $oprs(i)$ is $(init,t_1)\dots(init,t_{d_0}),(i^{j_1},t_{d_0+1}),\dots,(i^{j_1},t_{d_0+d_1}),\dots$. We know the dates t_k because of the clock calculus and the values because of the data dependency word w. If we apply the Kahn's semantics on this flow for the operator $\circledast\hat{}\,k$, we have:

   $$\circledast\hat{}^{\#}((init,t_1)\dots(init,t_{d_0}),(i^{j_1},t_{d_0+1}),\dots,(i^{j_1},t_{d_0+d_1}),\dots,k) = \prod_{i=0}^{k-1}(init,t_1^i)\dots\prod_{i=0}^{k-1}(init,t_{d_0}^i).\prod_{i=0}^{k-1}(i^{j_1},t_{d_0+1}^i)\dots\prod_{i=0}^{k-1}(i^{j_1},t_{d_0+d_1}^i).\circledast\hat{}^{\#}(\dots,k)$$

   This entails that each data is consumed k times more than for w. Thus $l(w') = l(w) \leq \prod_{(/\hat{}\,k_i)\in oprs} k_i$ by induction hypothesis. Furthermore, $nd(w') = nd(w)\cdot k$;
3. if $op = /\hat{}\,k$, then the computation of w' is a little bit tricky. We must remove $k-1$ consumptions every k in w. This obviously decreases d_j but this may also imply that some instances are not consumed anymore, thus changing some j_l. We must first compute the new prefix $(-1,\lceil d_0/k\rceil)(j_1',d_1')$. If $d_0 \ mod \ k = 0$ then $j_1' = j_1$ and $d_1' = \lceil d_1/k\rceil$. Otherwise $j_1' = \Sigma_{l\leq p}\ j_l$ with $p = min_j\{(d_0 \ mod \ k + \Sigma_{l\leq j}\ d_l) \geq k\}$ and $d_1' = \lceil (d_0 + \Sigma_{l\leq p}\ j_l - k)/k\rceil$. Indeed, by an Euclidean division, $d_0 = k\cdot q + r$ with $q,r\in\mathbb{N}$ and $r < k$. The flow $oprs(i)$ is $(init,t_1)\dots(init,t_k)\dots(init,t_{k\cdot q})\dots(init,t_{k\cdot q+r}),(i^{j_1},t_{d_0+1}),\dots,(i^{j_1},t_{d_0+d_1}),\dots$. If we apply the Kahn's semantics on this flow for the operator $/\hat{}\,k$, we have:

   $$/\hat{}^{\#}(1,(init,t_1)\dots(init,t_k)\dots(init,t_{k\cdot q})\dots(init,t_{k\cdot q+r})(i^{j_1},t_{d_0+1})\dots,(i^{j_1},t_{d_0+d_1})\dots,k) = (init,t_1).(init,t_k)\dots(init,t_{k\cdot q})./\hat{}^{\#}(r,(i^{j_1},t_{d_0+1})\dots,(i^{j_1},t_{d_0+d_1})\dots,k)$$

   We then have to copy r times the sub-word $(j_{p+1},d_{p+1})\dots(j_{p+l(w)-1},d_{p+l(w)-1})$ into the word w_i such that $nd(w_i)$ is divisible by k. We take a consumption every k times on the word w_i and we obtain the word w' such that $l(w') \leq l(w_i)$ and $nd(w') = nd(w_i)/k$. We have the length $l(w_i) = lcm(nd(w),k)/nd(w)\cdot l(w) \leq k\cdot l(w) \leq k\cdot\prod_{(/\hat{}\,k_i)\in oprs} k_i$.
   To compute the word w', we make an argument similar to the calculation of the prefix: we move forward in the word w_i as long as $\Sigma_j d_j \leq k$ and we keep the instances where this sum is $> k$. We then have $j' = j_{prec} + \Sigma_l\ j_l$ and $d' = \lceil (d + \Sigma_l\ d_l + (d_{prec} \ mod \ k) - k)\rceil$. We continue the move by resetting $\Sigma_l\ d_l$ and $\Sigma_l\ j_l$. Note also that $d_n = d_1$, because it is the repetition of the first instance. This ensures the periodicity of the calculus.

Data dependency words provide a factorised representation of data dependency between expression and therefore a compact representation of tasks precedences. They are the basis for our buffering communication protocol.

3.2 Task Extraction

The PRELUDE compiler first checks the correctness of the program to compile thanks to a series of static analyses: a standard type-checking (using classic techniques described in Pierce (2002)), a causality analysis similar to that of LUSTRE (see Halbwachs et al (1991b)) and more importantly a *clock calculus* (defined in Forget et al (2008)) that computes a clock for every expression of the program, from which we deduce the period of the expression and therefore the period of each imported node (i.e. task). Once static analyses succeed, the compiler can translate the program into a set of tasks.

We consider a classical task model derived from the fundamental work of Liu and Layland (1973). A task τ_i has a set of real-time attributes (T_i, C_i, D_i). τ_i is instantiated periodically with period T_i. For all $p \in \mathbb{N}^\star$, $\tau_i[p]$ denotes the p^{th} iteration of task τ_i. C_i is the worst case execution time (WCET) of the task. D_i is the relative deadline of the task. For all $p \in \mathbb{N}^*$, let $d_i[p]$ denote the absolute deadline of the instance p of τ_i, we have $d_i[p] = (p-1)T_i + D_i$. These definitions are illustrated in Figure 6. The restriction of PRELUDE we consider in this paper does not include phase offset operators, thus offsets (i.e. the release time of the first instance of a task) are all equal to zero.

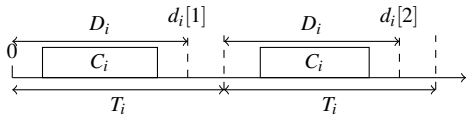

Fig. 6 Real-time characteristics of task τ_i

Each imported node call of the program is translated into a task. Each input/output of the main node of the program is also translated into a task (sensor/actuator task). The real-time attributes of each task are computed as follows:

- the WCET is specified by the corresponding imported node declaration;
- the period is defined by the clock of the task, which is computed by the clock calculus;
- by default, the relative deadline of a task is its period. Deadline constraints can also be specified (eg `o: due n`), then the deadline of the task is the minimum of the deadlines of its outputs.

3.3 Dependencies between Tasks

The program - a set of equations - is not transformed into a simple task set, but instead into a *task graph* where tasks are related by dependencies. The generated graph (V, E) consists of a set of nodes V corresponding to the tasks extracted as detailed in the previous section and of a set of edges E corresponding to task dependencies. Each dependence $e \in E \subseteq V \times V$ is labeled by the list of rate transition operators applied between the two communicating tasks.

Dependencies imply precedences, and thus define a partial order for the execution of the tasks, as well as detailed communications patterns. For instance, if an imported node A is applied to an output flow of another imported node B then the node B must execute before A. There are two types of communications: *direct* when the label of the dependence is a sequence of operators $*\hat{}k$, $/\hat{}$ or a direct call; *indirect* when there are some `fby` .

The task graph for the flight control is given in the Figure 7. It is a fairly straightforward translation of the program given in Section 2.4.

Proposition 2 *A dependence $\tau_1 \rightarrow^{oprs} \tau_2$ can be represented by the data dependency word $w = (-1, d_0)(k_1, d_1) \ldots (k_n, d_n)$ associated to oprs. Indeed, we have*

$$\forall q, l \in \mathbb{N}^2, l \in [1,n], \tau_1[q \cdot (\Sigma_{j \in [2,n]}\, i_j) + \Sigma_{j \leq l}\, i_j] \rightarrow \tau_2[q \cdot (\Sigma_{j \in [2,n]}\, d_j) + \Sigma_{j \leq l}\, d_j]$$

Proof $\tau_1 \rightarrow^{oprs} \tau_2$ is the precedence resulting from an expression of the form $\tau_2(\ldots, oprs(\tau_1(\ldots)), \ldots)$. Thus our proposition is a straightforward consequence of Proposition 1.

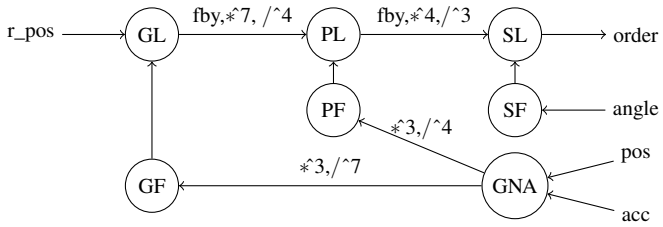

Fig. 7 Task graph of the Flight Control System

Example 3 Let us illustrate the computation of data dependency words on the flight control example. Let us consider the precedence $GNA \rightarrow^{*\hat{}3,/\hat{}4} PF$. It can be represented by the word $(-1,0)(1,1)(1,1)(1,1)(2,1)$. Indeed, as shown in Example 2, the data dependency word associated to the operator $*\hat{}3$ is $w_1 = (-1,0)(1,3)(1,3)$. There is no initial value, so the prefix for $/\hat{}4(w_1)$ is $(-1,0)(1,\lceil 3/4 \rceil) = (-1,0)(1,1)$.

Then, $4-3=1$ consumer instance needs to be removed. We have $nd(w_1) = 3$. We copy 4 times the pattern $(1,3)$. We obtain $w_i = (1,3)(1,3)(1,3)\ (1,3)$. In $w_i[1] = (1,3)$, 1 among 3 is used to complete the required instance from the prefix. Then $/\hat{}4(w_i[1]) = (1,\lceil 2/4 \rceil) = (1,1)$ and $4-2$ consumer instances need to be removed. In $w_i[2] = (1,3)$ we remove the missing 2 and $/\hat{}4(w_i[2]) = (1,\lceil 1/4 \rceil) = (1,1)$. 3 remain, thus all the consumer instances of $w_i[3] = (1,3)$ are used to complete the computation. $/\hat{}4(w_i[3]) = (2,\lceil 3/4 \rceil)$. And we find the same state as the one from the prefix. To summarize, the dependent task set associated to the flight control system is:

Tasks	**Dependencies**	**Data dependency words**
GNA=(30,5,30)	$GNA \rightarrow^{*\hat{}3,/\hat{}4} PF$	$(-1,0)(1,1)(1,1)(1,1)(2,1)$
SF=(30,5,30)	$GNA \rightarrow^{*\hat{}3,/\hat{}7} GF$	$(-1,0)(1,1)(2,1)(2,1)(3,1)$
SL=(30,5,30)	$SF \rightarrow SL$	$(-1,0)(1,1)(1,1)$
PF=(40,5,40)	$PF \rightarrow PL$	$(-1,0)(1,1)(1,1)$
PL=(40,5,40)	$GF \rightarrow GL$	$(-1,0)(1,1)(1,1)$
GF=(70,7,70)	$GL \rightarrow^{\texttt{fby},*\hat{}7,/\hat{}4} PL$	$(-1,2)(1,2)(1,2)(1,1)(1,2)(1,2)$
GL=(70,7,70)	$PL \rightarrow^{\texttt{fby},*\hat{}4,/\hat{}3} SL$	$(-1,2)(1,1)(1,1)(1,2)(1,1)$

4 Multitask Implementation of the System

We present in this section the multitask implementation of a PRELUDE program. The compiler has generated a set of dependent real-time tasks whose precedences are expressed by a set of data dependency words. The objective is then to generate a multithreaded implementation of the tasks that can be managed by an on-line scheduler such that the execution respects the semantics of the PRELUDE program. This requires two steps:

1. **ensure the respect of the precedences:** As we briefly mentioned in the introduction, there are mainly two different approaches available for handling precedences. The first approach relies on the use of (binary) semaphore synchronizations: a semaphore is allocated for each precedence and the destination task of the precedence must wait for the source task of the precedence to release the semaphore before it can start its execution. In the second approach, the respect of precedence constraints is simply ensured by the

way priorities and release times are assigned to tasks. We are interested in the second approach, because it provides necessary and sufficient feasibility conditions (while only sufficient conditions are available for the first approach) and does not require to certify semaphore synchronization mechanisms;

2. **provide a semantics-preserving communication protocol:** The respect of the precedences is not sufficient to ensure the semantics preservation. As an example, if the first instance of a task τ_1 consumes the data produced by the first instance of a quicker task τ_2, it may that in a multithreaded execution, the first instance of τ_1 occurs after the second instance of τ_2. Thus, if we do not store the data, τ_1 will access to the data produced by $\tau_2[2]$, that does not correspond to the semantics. A communication protocol is thus required for the multithreaded execution to respect the initial semantics of the program. We show how to compute this optimal protocol and how to generate a static wrapping code for each task. This wrapping code details when and where a task must write and read its data.

4.1 Precedence Encoding

We propose to encode precedences in task real-time attributes. Chetto et al (1990) have defined an encoding that transforms a dependent task set into an equivalent independent task set. The independent task set is then scheduled with the classic Earliest-Deadline-First (EDF) policy. This approach (encoding+EDF) is optimal in the class of dynamic-priority schedulers (where priorities are assigned at run-time), in the sense that if there exists a dynamic priority assignment that respects all task real-time attributes and precedences, this approach finds one such assignment. This work only supports precedences between non-repeated *jobs* (a periodic task is a periodic repetition of jobs) but can easily be extended to precedences between tasks of the same period (called simple precedences). Using a similar approach, Forget et al (2010) have defined optimal encodings for several static priority policies (where priorities are assigned before run-time), including deadline monotonic (DM). In both cases (static or dynamic priority), the encoding consists in:

1. adjusting the release dates so that for any precedence $\tau_1 \rightarrow \tau_2$, the start time of τ_2 is always after the start time of τ_1;
2. assigning priorities so that for any precedence $\tau_1 \rightarrow \tau_2$, the task τ_1 has a higher priority than τ_2.

In our context all the offsets are equal to zero, thus we do not need to adjust release dates. Furthermore, precedences can relate two tasks with different periods, in which case only a subset of the instances of the two tasks are related by a precedence constraint. The assignment of task priorities for both DM (static priority) and EDF (dynamic priority) is based on task deadlines, thus to ensure that priorities are assigned to tasks in a way that respects precedences, we simply have to adjust task deadlines. The precedence encoding technique is formalized as follows:

Proposition 3 *The dependent task set* $\{\tau_1 = (T_1, C_1, D_1), \ldots, \tau_p = (T_p, C_p, D_p)\}$ *constrained by a set of precedences expressed by a set of data dependency words* $\{w_{i,j}\}_{i,j \in P \subseteq [1,p]^2}$ *is equivalent to the independent task set* $\{\tau_1 = (T_1, C_1, D_1^\star), \ldots, \tau_n = (T_p, C_p, D_p^\star)\}$ *whose data consumption is described by the set of data dependency words* $\{w_{i,j}\}_{i,j \in P \subseteq [1,p]^2}$ *where*

$$D^\star(\tau_i) = min_{j \mid w_{i,j} = (-1,d_0)(i_1,d_1)\ldots(i_n,d_n)} \; min_{k \leq n} \\ \left\{ D^\star(\tau_j) + (\Sigma_{l \leq k-1} d_l + 1) T(\tau_j) - \; max(0, (\Sigma_{l \leq k} i_l - 1)) T(\tau_i) - C(\tau_j) \right\}$$

Proof The algorithm (Chetto et al (1990); Forget et al (2010)) modifies the absolute deadline of a task τ to be $d^\star = min_{j \mid \tau \to \tau_j}(d, d_j^\star - C_j)$. In our context, we do not only consider simple precedences $\tau \to \tau_j$, which correspond to $oprs = []$, but more general precedences. Let $\tau_1 \to^{oprs} \tau_2$, we compute the associated dependency word $w = (-1,d_0)(i_1,d_1)\ldots(i_n,d_n)$. We deduce all the constraints on the deadlines of τ_1. The instance $\tau_1[\Sigma_{j\leq k} i_j]$ must end before the beginning of the first instance of τ_2 that consumes the data produced by $\tau_1[\Sigma_{j\leq k} i_j]$, that is $\tau_2[\Sigma_{\leq k-1} d_j + 1]$ (by definition of the dependency word). This provides the absolute constraints for $k = 1,\ldots,n$: $d^\star(\tau_1[\Sigma_{j\leq k} i_j]) \leq d(\tau_2^\star[\Sigma_{j\leq k-1} d_j + 1]) - C(\tau_2)$. As we manipulate relative deadlines, we adapt this formula by modifying the relative deadlines. Note that it is sufficient to reason on the first n instances because of the periodic pattern of the word w. Hence:

$$D^\star(\tau_1) \leq min_{k\leq n}\left\{D^\star(\tau_2) + (\Sigma_{j\leq k-1} d_j + 1)T(\tau_2) - max(0, (\Sigma_{j\leq k} i_j - 1))T(\tau_1) - C(\tau_2)\right\}$$

We deduce the relative deadline of τ_1 to be the greatest value respecting these constraints. Because of the hypothesis on the delays (`fby` at the beginning of the operators list), there is no loop in the constraints.

Example 4 The task set for the simplified flight control becomes after encoding $SL = (30,5,30)$, $SF = (30,5,25)$, $GNA = (30,5,30)$, $PF = (40,5,35)$, $PL = (40,5,40)$, $GF = (70,7,63)$ and $GL = (70,7,70)$.

To summarize, our scheduling algorithm works as follows:

1. Encode precedences using Proposition 3;
2. Perform a schedulability analysis. As encoded tasks are independent, we can reuse classic schedulability tests Cottet et al (2002). For instance, we used Cheddar [Singhoff et al (2004)] to verify that our previous example is schedulable for EDF but not schedulable for DM.
3. Schedule tasks either using DM (static priority) or EDF (dynamic priority).

4.2 Communication protocol

Problems due to the Interleaving In order to preserve the PRELUDE semantics, consuming task instances must use data produced by the correct producing task instance. Due to the multi-rate aspects of the systems, precedence encoding is not sufficient and additional communication mechanisms are required.

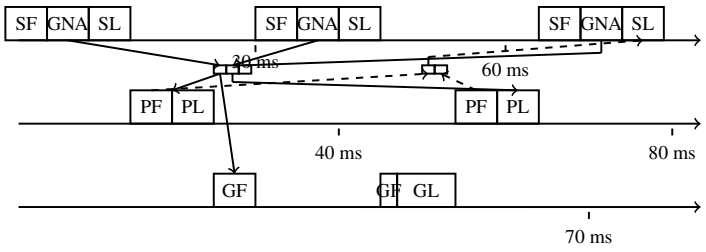

Fig. 8 Example of an execution

To illustrate this problem, let us consider data produced by *GNA* in the flight control example, which is consumed by *PF* and *GF*. Figure 8 shows that data produced by the first instance of *GNA* must be stored over several periods. We have two choices: either we consider that data must remain available over the complete period of *GF* that is 70 ms, or we decide that data must remain available for the worst response time of the consumers. We choose the first solution because the second solution is not necessarily efficient and it is more complex. The task *SL* requires data produced by *PL* with a delay. This requires two cells to store the data produced by *PL*: one for the previous value and one for the current.

Sofronis et al. *Method* The authors of Sofronis et al (2006) consider a real-time task set obtained from a LUSTRE program (the construction of the task set from the code is assumed to be out of the scope of their work). The authors first study the mechanism to implement a communication protocol based on the use of buffers. Then, they propose to optimise the number of buffers required for multi-periodic tasks. Their algorithm consider a set of communications $\tau \to \tau_1, \ldots, \tau \to \tau_n$ and for each data produced during the hyperperiod:

1. test whether data is consumed or not;
2. if it is the case, test if it possible to reuse an existing buffer cell to store the data. If no cell can be reused, add a new buffer cell and increase the number of cells by one.

Communication Protocol In order to apply this algorithm on the communications described in the precedence graph, we must determine whether a cell can be reused or not. A cell can be reused at date t, if the previous stored data was used until at most $t' \leq t$. This information is already stored within the data dependency words. The lemma gives the formula:

Lemma 1 *Let us consider the precedence $\tau_1 \to^{oprs} \tau_2$ represented as a data dependency word $w_{1,2} = (-1,d_0)(i_1,d_1)(i_2,d_2)\ldots(i_n,d_n)$. The instance h of τ_1 is consumed iff $\exists q,l \in \mathbb{N}^2$, $l \in [1,n]$ such that $h = q \cdot (\Sigma_{j\in[2,n]}\, i_j) + \Sigma_{j\leq l}\, i_j$. The maximal date of use of instance $h = q \cdot (\Sigma_{j\in[2,n]}\, i_j) + \Sigma_{j\leq l}\, i_j$ is*

$$T(\tau_2) \cdot (q \cdot (\Sigma_{j\in[2,n]}\, d_j) + \Sigma_{j\leq l}\, d_j - 1) + D(\tau_2)$$

Proof The lemma is a simple transcription of the definition of data dependency words. The first instance to be consumed i_1, is consumed by the jobs $d_0+1,\ldots,d_0+d_1$ of τ_2. The data must be available till the end of the execution of the last job to consume it, which is $T(\tau_2)\cdot(d_0+d_1-1)+D(\tau_2)$. This can be generalised to any consumed instance.

Example 5 The simple precedence *SF* $\to$ *SL* is coded by $(-1,0)(1,1)(1,1)$. Each data must be available exactly one period of 30.

GNA $\to^{*3,/\hat{}4}$ *PF* is coded by $(-1,0)(1,1)(1,1)(1,1)(2,1)$. The first third data must be available 40 ms, the fourth is not consumed and the fifth must be available 40 ms as well. The scheme repeats following the data dependency word definition.

For each task, we add a buffer where the task writes the data that will be consumed by some other task. We can now apply the algorithm for using a minimal number of cells for each of these buffers.

Proposition 4 *The task set $\{\tau_1 = (T_1,C_1,D_1),\ldots,\tau_p = (T_p,C_p,D_p)\}$ whose data consumption is described by a set of data dependency words $\{w_{i,j} = (-1,d_0^{i,j})(i_1^{i,j},d_1^{i,j})(i_2^{i,j},d_2^{i,j})\ldots(i_{n^{i,j}}^{i,j},d_{n^{i,j}}^{i,j})\}_{i,j\in P\subseteq[1,p]^2}$ is equivalent to the task set $\{\tau_1 = (T_1,C_1,D_1),\ldots,\tau_p = (T_p,C_p,D_p)\}$*

whose data consumption is described by the communication vectors $\{V_1,\ldots,V_p\}$ *where* $V_k[h]=[b,L_k]$ *with*

$$\begin{cases} h\in[1,lcm_j(\Sigma_{l\leq n_{k,j}} i_l^{k,j})] \\ b \text{ is the buffer cell where the data } h \text{ is stored} \\ L_k = concat\{(\tau_j, q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, d_l^{k,j})+\Sigma_{l\leq m-1}\, d_l^{k,j}-1,\ldots,q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, d_l^{k,j})+\Sigma_{l\leq m}\, d_l^{k,j} \\ \qquad\qquad \mid q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, i_l^{k,j})+\Sigma_{l\leq m}\, i_l^{k,j}=h\} \end{cases}$$

Proof For all the consumptions of τ_1, $\tau_1 \rightarrow^{oprs} \tau_j$, $j=2,\ldots,m$, we have the m data dependency words $w_{1,j}=(-1,d_0^{1,j})(i_1^{1,j},d_1^{1,j})(i_2^{1,j},d_2^{1,j})\ldots(i_{n^{1,j}}^{1,j},d_{n^{1,j}}^{1,j})$. The data dependency words describe the communication on different instances and each pattern restarts at some point, which is not necessary common. We have to reason on the hyperperiod of these instances, that is $lcm_j(\Sigma_{l\leq n_{1,j}} i_l^{1,j})$. We unfold the data dependency words so that we describe the consumptions until the *lcm*. The communication is represented by a vector of length *lcm*. Each element of the vector $V[h]$ describes for the data $h\leq lcm_j(\Sigma_{l\leq n_{1,j}} i_l^{1,j})$ the buffer cell where it is stored and the list of consumers and instance numbers.

The list of consumers can directly be found from the unfolded data dependency words. Indeed h is consumed by τ_j if and only if $\exists q,m$ such that $q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, i_l^{k,j})+\Sigma_{l\leq m}\, i_l^{k,j}=h$. If h is consumed, we know exactly the consumer jobs which are $q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, d_l^{k,j})+\Sigma_{l\leq m-1}\, d_l^{k,j}-1,\ldots,q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, d_l^{k,j})+\Sigma_{l\leq m}\, d_l^{k,j}$. The last date of consumption of data h is a direct application of lemma 1. We take the maximal consumption date of each consumer.

The value of b can be written as an algorithm. Let us denote by *nb* the number of cells of the buffer associated to τ_1. Let us instantiate $nb=0$. For all h, we test if h is consumed which is equivalent to $\exists j,q,m$ such that $q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, i_l^{k,j})+\Sigma_{l\leq m}\, i_l^{k,j}=h$. Then

1. if h is not consumed, $V_1[h]=(0,[])$;
2. if h is the first data to be consumed, i.e. $h=min(i_1^{1,j})$. Then h is stored in the first cell. Thus $nb=1$ and $V_1[h]=(1,\ldots)$. The cell is locked within the interval $[(h-1)\cdot T(\tau_1),d]$ where $d=max\{T(\tau_j)\cdot(q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, d_l^{k,j})+\Sigma_{l\leq m}\, d_m-1)+D(\tau_j)\mid q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, i_l^{k,j})+\Sigma_{l\leq m}\, i_l^{k,j}=h\}$;
3. if h is not the first data to be consumed, we test if a buffer $b\in[1,nb]$ is available in the interval $[(h-1)\cdot T(\tau_1),d]$ where $d=max\{T(\tau_j)\cdot(q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, d_l^{k,j})+\Sigma_{l\leq m}\, d_m-1)+D(\tau_j)\mid q\cdot(\Sigma_{l\in[2,n_{k,j}]}\, i_l^{k,j})+\Sigma_{l\leq m}\, i_l^{k,j}=h\}$. If so, we reuse the buffer b, otherwise we increment $nb=nb+1$ and use the buffer $b=nb$.

Example 6 Let consider the buffer associated to *GNA*, it requires 3 cells. There two readers $GNA \rightarrow^{*3,/4} PF$ and $GNA \rightarrow^{*3,/7} GF$. The dependency words are respectively $(-1,0)$ $(1,1)(1,1)(1,1)(2,1)$ and $(-1,0)(1,1)(2,1)(2,1)(3,1)$. We have $lcm(\sum(1+1+1+2),\sum(1+2+2+3))=lcm(5,8)=40$. The vector is described until the instance 40 of *GNA*.

The first instance of *GNA* is consumed by the first instances of *PF* and *GF*. The last date of consumption is $max\{40,63\}=63$. Hence, $V[1]=(1,[(PF,1);(GF,1)])$. The second instance $GNA[2]$ is only consumed by *PF* between $[30,65]$, hence $V[2]=(2,[(PF,2)])$. The third instance is consumed by both *PF* and *GF* between $[60,126]$ Hence $V[3]=(3,[(PF,3);(GF,2)])$. And so on until V reaches the instance 40 of *GNA*.

Finally, the independent task set associated to the flight control system is:

Tasks	Number of cells	Communication vector
GNA=(30,5,30)	3	$V_{GNA} = [(1,[(PF,1);(GF,1)]);(2,[(PF,2)])\ldots]$
SF=(30,5,25)	1	$V_{SF} = [(1,[(SL,1)]);(2,[(SL,2)])]$
SL=(30,5,30)	0	
PF=(40,5,35)	1	$V_{PF} = [(1,[(PL,1)]);(2,[(PL,2)])]$
PL=(40,5,40)	3	$V_{PL} = [(1,[(SL,3)]);(2,[(SL,4,5)])\ldots]$
GF=(70,7,63)	1	$V_{GF} = [(1,[(GL,1)]);(2,[(GL,2)])]$
GL=(70,7,70)	3	$V_{GL} = [(1,[(PL,3,4)]);(2,[(PL,5,6)])\ldots]$

Code Generation From such vectors V, it is quite easy to deduce wrapping codes for each task, which describes in which cell data must be read or written. In contrast to Sofronis et al (2006), we do not use pointers but instead a static description of the buffers to access for each instance of the tasks.

Example 7 We explain an extract of the code generated for Flight Control System for the data produced by `GNA`. The initial function is wrapped by a code for the communication protocol:

1. Communication buffers are declared as global variables (here `GNA_o[3]` for the 3 cells);
2. A function pointer is generated for each task (here `GNA_fun`). It corresponds to the body of the task, executed by each instance of the task. The structure of each function is as follows:
   (a) Communication protocols are declared. In this example, `write_pat` specifies when `GNA` must write in the communication buffer;
   (b) The function calls the external function provided by the user for the task (i.e. the function describing the functional behaviour of the task);
   (c) Outputs are copied to the communication buffer according to the "write pattern".

```
int GNA_o[3]; // buffer for GNA -> GF, 3 cells
void* GNA_fun(void* args) //wrap the external function GNA with com protocol
{
  struct write_proto_t o_write; // vector V, tells when GNA must write
  o_write.write_pat[0] = 1; o_write.write_pat[1] = 1;
  ...
  GNA(pos_GNA,acc_GNA,&GNA_fun_outs); // call to the external GNA function
  if(o_write.write_pat[instance]) { // test if data must be stored
    GNA_o[o_wcell]=GNA_fun_outs.o; // write in the communication buffer
    o_wcell=(o_wcell+1)%3; // update cell index
  }
  ...
    (instance++)%40; // update number of instances
}
```

5 Experimental results

The theoretical results have been implemented in a compiler prototype. It takes a PRELUDE program and produces C code. The approach, and the prototype, have been applied on several examples and two representative case studies which are detailed below.

5.1 Prototype implementation

A prototype of the compiler has been implemented in OCAML (Leroy (2006)) and is available for download[3]. It supports the language defined in Sec. 2 and implements the complete compilation chain, with proper error handling, in approximately 4000 lines of code. The prototype generates C code independent from the target OS.

The sequence of computations performed by the compiler is given below and the number of lines of code for each part of the compiler is given in Fig. 9).

Syntax analysis	300
Typing	400
Clock calculus and clock data types	1000
Causality analysis	100
Task graph extraction	600
Precedence encoding	400
Communication protocol	200
C code generation	400
Base data types and utility functions	600
Command line processing and main file	100
Total	4100

Fig. 9 The different parts of the prototype

For a given PRELUDE program, the user can obtain several outputs:

1. the first result is whether the static analyses, in particular the clock calculus, succeed or not: this ensures that the semantics of the program is well defined. The result of the typing and of the clock calculus can also be printed, which is mostly useful in case the user left the clock/type of some inputs/outputs unspecified. He/she can verify that the result corresponds to what he expected;
2. a Cheddar [Singhoff et al (2004)] model of the independent task set. This way, he/she can verify the schedulability of the program;
3. the generated C code.

The generated C code is independent of the target OS. A simple integration code, specific to the target OS (but generic for the different systems written by the programmer), must be written. This integration code must declare one thread for each task described by the generated C code and run all the threads concurrently.

The distribution of the compiler provides all the code required to compile and execute the generated C code (generic integration code+EDF scheduler) using MARTE OS (Rivas and Harbour (2002)). This Operating System was designed to ease the implementation of application-specific schedulers while remaining close to the POSIX extensions for real-time (POSIX.13 (1998)). This RTOS also has the advantage of being executable on top of a standard LINUX station, which simplifies the debugging and test phases. We are currently working on the support of a standard C POSIX OS.

5.2 Case Study 1: Partial ATV Navigation Function

The full description of this case study can be found in Forget (2009).

[3] `http://www.lifl.fr/~forget/prelude.html`

General Presentation of the FAS The control system of the Automated Transfer Vehicle has been developed by EADS Astrium Space Transportation. The control system is made up of two parts: the Flight Application Software (FAS) and the Mission Safing Unit (MSU). The FAS is the main part of the system, it handles all the software functionalities of the system as long as no fault is detected. The MSU watches the FAS to detect if it is faulty. If a fault occurs, the MSU disables the FAS and starts a safety procedure, which stops the current ATV actions and moves it to a safe orbit, with respect to the station, waiting for further instructions from the ground station.

When applying our approach, the designer does not need to compute these three sequences statically. We have described the behaviour of the system at the service level and generated the corresponding multithreaded code.

Current implementation The software is composed of 240 periodic operations, also called services. There are also sporadic services which are not considered in the following. In the current implementation by EADS Astrium Space Transportation, the services are first been developped separately in ADA (but could also be developped using LUSTRE for instance). Their integration into the global multi-periodic system is then mostly done by hand. The numerous services are manually distributed between 3 tasks of frequencies 0.1Hz, 1Hz and 10Hz. The services inside each task have then been sequenced manually. The 10Hz task does not only contain services activated with a frequency of 10Hz, it also contains slower services, for instance services with a frequency of 1Hz, which are activated only one out of 10 successive instances of the 10Hz task. A RM scheduler is then used for to execute the tasks concurrently.

Prelude programming Our implementation is mainly based on two design documents provided by EADS Astrium Space Transportation. The first one specifies the real-time constraints of the different services of the FAS. We kept 180 services out of the 240 periodic services specified in this document, discarding services for which neither bus transfers nor precedences with other services were specified. The main node has 70 inputs and 9 outputs. The complete program is about 500 lines of code long. On a current PC (Intel Code 2 Duo @ 3GHz with 2Go of RAM), the compilation of the program takes about 50ms to complete. The code generated for the program is about 8000 lines long.

The generated task set was proved to be schedulable using CHEDDAR. We also ran a simulation of the system with MARTE OS. This case study proves the *feasibility* of our approach. Compared to the current implementation, our implementation has two importants benefits. First, the integration level can be specified formally. Second, it completely avoids the tedious and time-consuming manual scheduling of services.

5.3 Case Study 2: Avionics Function

The second case study is an avionics function specification extracted from Boniol et al (2008). The authors take as an input a partial specification, meaning the data-flow between the tasks are not fully expressed, and compute a multiprocessor implementation respecting the initial constraints. Since the initial specification is not complete, there are several PRELUDE programs that respond to the constraints, as there were several multiprocessor allocations.

The application is made up of 762 tasks, 4 periods (10, 20, 40 and 120ms) and 236 *imposed* simple precedences. We have imposed a choice of data-flow for the communication

which were not precised. This leads to 1415 precedences. The compilation of the program takes about 1s.

From this specification, we derived a larger example, where the PRELUDE program is generated by a semi-random generator. The PRELUDE program consists of about 3000 tasks and is over 1Mo large. The compilation takes about 4.5s.

This shows the *scalability* of our approach, as we can see that programs larger than real-life examples can easily be supported by the PRELUDE compiler.

6 Conclusion

We described a complete code generation process, which starts from the description of the real-time software architecture of a multi-periodic systems and translates it into a multi-threaded code executable with an on-line priority based scheduler. The framework is dedicated to the integration of several locally mono-periodic functions developed separately into a globally multi-periodic system. We first showed the feasibility of the approach and then showed that we can automatically generate an implementation that respects both the functional and the temporal semantics of the original program.

When studying the real case study, we noticed that the granularity of the *imported nodes*, which are each translated into a separate real-time task, is not adapted to an actual RTOS. Indeed, most RTOS accept at most one hundred tasks, while our case study produces 750 tasks. In the future, we will try to find some clustering techniques to group several imported nodes into the same task, in order to reduce the overall number of tasks.

A second conclusion when applying PRELUDE to the case study is that an additional layer of abstraction might still be required on top of PRELUDE. Indeed, combinations of rate transition operators are not always easy to write. Thus, we are planning to develop a graphical editor in a framework such as TopCased, which would provide automated communication pattern generation (ie generation of sequences of rate transition operators).

References

ARINC (2005) ARINC Specification 653: Avionics Application Software Standard Interface. Aeronautical Radio INC

Aubry P, Le Guernic P, Machard S (1996) Synchronous distribution of Signal programs. In: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture, pp 656–665

Baruah SK, Burns A (2006) Sustainable scheduling analysis. In: Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06), IEEE Computer Society, pp 159–168

Benveniste A, Le Guernic P, Jacquemot C (1991) Synchronous programming with events and relations: the Signal language and its semantics. Science of Computer Programming 16(2):103–149

Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages 12 years later. Proceedings of the IEEE 91(1):64–83

Boniol F, Hladik PE, Pagetti C, Aspro F, Jégu V (2008) A framework for distributing real-time functions. In: Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'08), Springer, Lecture Notes in Computer Science, vol 5215, pp 155–169

Cervin A (2003) Integrated control and real-time scheduling. PhD thesis, Department of Automatic Control, Lund University, Sweden

Chetto H, Silly M, Bouchentouf T (1990) Dynamic scheduling of real-time tasks under precedence constraints. Real-Time Systems 2(3):181–194

Cohen A, Duranton M, Eisenbeis C, Pagetti C, Plateau F, Pouzet M (2006) *N*-Synchronous Kahn Networks: a relaxed model of synchrony for real-time systems. In: ACM International Conference on Principles of Programming Languages (POPL'06), Charleston, USA

Colaço JL, Pouzet M (2003) Clocks as first class abstract types. In: Proceedings of the 3rd International Conference on Embedded Software (EMSOFT'03), Philadelphia, USA, Lecture Notes in Computer Science, vol 2855, pp 134–155

Cottet F, Delacroix J, Kaiser C, Mammeri Z (2002) Scheduling in real-time systems. John Wiley & Sons

Curic A (2005) Implementing Lustre programs on distributed platforms with real-time constraints. PhD thesis, Université Joseph Fourier, Grenoble

Faucou S, Déplanche AM, Trinquet Y (2004) An ADL centric approach for the formal design of real-time systems. In: Architecture Description Language Workshop at IFIP World Computer Congress (WADL'04), vol 176, pp 67–82

Feiler PH, Gluch DP, Hudak JJ (2006) The architecture analysis & design language (AADL): An introduction. Tech. Rep. CMU/SEI-2006-TN-011, Carnegie Mellon University

Forget J (2009) A synchronous language for critical embedded systems with multiple real-time constraints. PhD thesis, Université de Toulouse - ISAE/ONERA, Toulouse, France

Forget J, Boniol F, Lesens D, Pagetti C (2008) A multi-periodic synchronous data-flow language. In: Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE'08), IEEE Computer Society, Nanjing, China, pp 251–260

Forget J, Boniol F, Grolleau E, Lesens D, Pagetti C (2010) Scheduling dependent periodic tasks without synchronization mechanisms (submitted). In: Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Stockholm, Sweden, pp 301–310

Girault A, Nicollin X (2003) Clock-driven automatic distribution of Lustre programs. In: Proceedings of the 3rd International Conference on Embedded Software (EMSOFT'03), Philadelphia, USA, Lecture Notes in Computer Science, vol 2855, pp 206–222

Girault A, Nicollin X, Pouzet M (2006) Automatic rate desynchronization of embedded reactive programs. ACM Trans Embedded Comput Syst 5(3):687–717

Graham RL (1969) Bounds on multiprocessing timing anomalies. SIAM Journal on Applied Mathematics 17(2):416–429

Halbwachs N (1993) Synchronous programming of reactive systems. Kluwer Academic Publisher

Halbwachs N, Caspi P, Raymond P, Pilaud D (1991a) The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE 79(9):1305–1320

Halbwachs N, Raymond P, Ratel C (1991b) Generating efficient code from data-flow programs. In: Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP '91), Passau, Germany, pp 207–218

Henzinger TA, Horowitz B, Kirsch CM (2003) Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE 91(1):84–99

Kahn G (1974) The semantics of simple language for parallel programming. In: Proceedings of the International Federation for Information Processing (IFIP'74) Congress, New York, USA, pp 471–475

Lee EA, Sangiovanni-Vincentelli AL (1996) Comparing models of computation. In: Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided design

(ICCAD'96), IEEE Computer Society, San Jose, USA, pp 234–241
Leroy X (2006) The Objective Caml system release 3.09, Documentation and user's manual. INRIA
Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM 20(1):46–61
OSEK (2003) OSEX/VDX Operating System Specification 2.2.1. OSEK Group, www.osek-vdx.org
Pierce BC (2002) Types and programming languages. MIT Press, Cambridge, USA
POSIX13 (1998) IEEE Std. 1003.13-1998. POSIX Realtime Application Support (AEP). The Institute of Electrical and Electronics Engineers
Pouzet M (2006) Lucid Synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI
Reynolds JC (1998) Theories of Programming Languages. Cambridge University Press
Rivas MA, Harbour MG (2002) POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In: Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS'02), Washington, USA, pp 67–75
Sename O, Simon D, Ben Gaïd MEM (2008) A LPV approach to control and real-time scheduling codesign: application to a robot-arm control. In: Proceedings of the 47th IEEE Conference on Decision and Control (CDC'08), Cancun Mexique, pp 4891–4897
Singhoff F, Legrand J, Nana L, Marcé L (2004) Cheddar: a flexible real time scheduling framework. Ada Lett XXIV(4):1–8
Smarandache I, Le Guernic P (1997) A canonical form for affine relations in signal. Tech. Rep. RR-3097, INRIA
Sofronis C, Tripakis S, Caspi P (2006) A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In: Proceedings of the 6th International Conference on Embedded Software (EMSOFT'06), Seoul, South Korea, pp 21–33
The Mathworks (2009) Simulink: User's Guide. The Mathworks
Tripakis S, Sofronis C, Caspi P, Curic A (2005) Translating discrete-time Simulink to Lustre. ACM Trans Embed Comput Syst 4(4):779–818